DOCUMENT RESUME

ED 149 731 IR 005 514

AUTHOR	Fife, Dennis W.
TITLE	Computer Software Management: A Primer for Project Management and Quality Control. Computer Science and Technology.
INSTITUTION	National Bureau of Standards (DCC), Washington, D.C. Inst. for Computer Sciences and Technology.
REPORT NO	NBS-SP-500-11
PUB DATE	Jul 77
NOTE	56p.; Some pages may be marginally legible due to print quality
AVAILABLE FROM	Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402 (C13.10:500-11, $2.00)
EDRS PRICE	MF-$0.83 HC-$3.50 Plus Postage.
DESCRIPTORS	*Administrator Guides; *Computer Oriented Programs; Management Systems; Managerial Occupations; *Material Development; *Program Administration; *Quality Control
IDENTIFIERS	*Computer Software

ABSTRACT

This brief guide is intended for managers who are responsible for computer projects, to explain the use of quality controls and software management methods. The typical problems of software development are explained. Over twenty distinct quality controls are defined, and recommendations are given for software management actions. Empirical information is included that would help top executives to appreciate the potential problems and importance of software management. (Author)

****
* Reproductions supplied by EDRS are the best that can be made *
* from the original document. *
****

ED149731

COMPUTER SCIENCE & TECHNOLOGY:

Computer Software Management: A Primer for Project Management and Quality Control

U.S. DEPARTMENT OF HEALTH, EDUCATION & WELFARE
NATIONAL INSTITUTE OF EDUCATION

THIS DOCUMENT HAS BEEN REPRODUCED EXACTLY AS RECEIVED FROM THE PERSON OR ORGANIZATION ORIGINATING IT. POINTS OF VIEW OR OPINIONS STATED DO NOT NECESSARILY REPRESENT OFFICIAL NATIONAL INSTITUTE OF EDUCATION POSITION OR POLICY

Dennis W. Fife

Systems and Software Division
Institute for Computer Science and Technology
National Bureau of Standards
Washington, D.C. 20234

IR005514

U.S. DEPARTMENT OF COMMERCE, Juanita M. Kreps, Secretary

Dr. Sidney Harman, Under Secretary

Jordan J. Baruch, Assistant Secretary for Science and Technology

NATIONAL BUREAU OF STANDARDS, Ernest Ambler, Acting Director

Issued July 1977

Reports on Computer Science and Technology

The National Bureau of Standards has a special responsibility within the Federal Government for computer science and technology activities. The programs of the NBS Institute for Computer Sciences and Technology are designed to provide ADP standards, guidelines, and technical advisory services to improve the effectiveness of computer utilization in the Federal sector, and to perform appropriate research and development efforts as foundation for such activities and programs. This publication series will report these NBS efforts to the Federal computer community as well as to interested specialists in the academic and private sectors. Those wishing to receive notices of publications in this series should complete and return the form at the end of this publication.

National Bureau of Standards Special Publication 500-11
Nat. Bur. Stand. (U.S.), Spec. Publ. 500-11, 58 pages (July 1977)
CODEN: XNBSAV

Library of Congress Cataloging in Publication Data

Fife, Dennis W.
Computer software management.
(Computer science & technology) (NBS special publication; 500-11)
Bibliography: p.
Supt. of Docs. No.: C13.10:500-11
1. Electronic data processing—Management. 2. Programming (Electronic computers)—Management. I. Title. II. Series. III. Series: United States. National Bureau of Standards. Special publication; 500-11.
QC100.U57 no. 500-11 [QA76.9.M3] 602'.1s [658'.05'4042] 77-608127

U.S. GOVERNMENT PRINTING OFFICE
WASHINGTON: 1977

For sale by the Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402 - Price $2
Stock No. 003-003-01795-7

PREFACE

The National Bureau of Standards is the primary technical advisor on computers for government agencies (Brooks Act, Public Law 89-306). NBS assists Federal agencies in specific projects, and so obtains first-hand experience with agency problems and competencies. This guide was prepared drawing upon extensive contacts with government computing personnel, and using the published recommendations of authorities in software management. A variety of executive guides and primers in other fields were reviewed in choosing the scope and approach to the material covered here. Other staff members of the Institute for Computer Sciences and Technology assisted by reviewing the material as it was developed. The author was helped particularly by the constructive comments of I. Trotter Hardy and Theodore Linden.

NOTICE TO READERS

This guide is intended to become a standard reference for managing Federal government software projects. Comments by the readers on its completeness, clarity, etc. would be valuable in making appropriate revisions for this purpose. A postage-free reader evaluation form is included at the back of the report, and you are encouraged to complete and return it.

TABLE OF CONTENTS

	Page
Why this guide was written	1
Who this guide is for	2
What this guide is for	2
What this guide is not for	3
How to use this guide	3
Why software management is important	3
Why every agency needs software management	5
#Basic tasks of software management	7
#The qualities of superior software	13
#Organizing for software management	15
#Minimizing early problems	21
#Guidelines for design and programming	29
#Guidelines for testing	33
#Avoiding endless software problems	38
#The need for software production tools	40
#Standards for quality control	42
#Fitting controls to the project size	45
Final homilies	47
References	48

#Section with specific guidelines.

List of Figures

		Page
1.	The Life Cycle of a Software Product.	10
2.	Chart of Basic Quality Control Documents . . .	11
3.	A Representative Team Organization	19
4.	A Representative Organization for a Large Project	20
5.	Chart of Recommended Minimum Milestones . . .	28
6.	Chart of Recommended Quality Controls	46

COMPUTER SOFTWARE MANAGEMENT:
A primer for project management and quality control

Dennis W. Fife

Today, providing computer software involves greater cost and risk than providing computer equipment, because hardware is mass produced by industry using proven technology, while software is still produced mostly by the craft of individual computer programmers. This brief guide is intended for managers who are responsible for computer projects, to explain the use of quality controls and software management methods. The typical problems of software development are explained. Over twenty distinct quality controls are defined, and recommendations are given for software management actions. Empirical information is included that would help top executives to appreciate the potential problems and importance of software management.

Keywords: computer management; computer programming; computer project control; computer software; software engineering; software quality; software reliability.

WHY THIS GUIDE WAS WRITTEN

Computer software is the computer instructions or programs, as well as the data files and descriptive manuals, that have to be provided to make computer equipment (hardware) perform useful services. Today, providing software involves greater cost and risk than providing the equipment, because hardware is mass produced by industry using proven technology, while software is still produced mostly by the craft of individual computer programmers and users.

Software management is the technical and management control throughout the life cycle of all software that supports an organization's mission and objectives. Software management is effective when high quality software is being obtained with reasonable and controllable costs. Software management is inadequate when poor decisions and lax technical controls result in faulty software performance, computer service fiascos, or project cost overruns. The unabated high cost of software as well as many publicized cases of software catastrophes are convincing evidence that software management is poorly understood and practiced in most organizations.

The problem is deep-seated and serious, because many managers who control software projects do not have professional computing expertise, and many programmers lack the technical skills to properly design and produce the highly complex computer software needed today. Equally important, the expansion of computer applications over the last twenty years has far outpaced the evolution of basic principles and engineering discipline in software design and production. Much more applied research is needed before software engineering will progress to the effectiveness found in other engineering and production fields.

WHO THIS GUIDE IS FOR

This guide is intended for managers and staff who are responsible for computer projects. These will be data processing professionals and supervisors primarily. But the intended audience also includes executives and managers who were not trained as computer professionals and who perhaps have never used a computer themselves. Every reader will need to understand the rudiments of computer operation and programming.

WHAT THIS GUIDE IS FOR

The objective of this guide is to give the reader an appreciation of the software pitfalls in computer projects, and to explain the use of quality controls and software management methods. The typical problems of software production are highlighted, and recommendations are given

for setting up software management procedures.

WHAT THIS GUIDE IS NOT FOR

The brevity and special purpose of this guide mean that it is unsuitable for other important purposes. In particular, this guide is NOT a survey of programming methods, NOR a handbook for the direct supervision of computer programmers. Its use cannot compensate for a lack of technical expertise that is essential to direct management of software design, development, and maintenance. Finally, this guide is NOT a popular treatment to arouse the interest and concern of the general public. It is expected that readers have a professional interest and obligation in decision-making for software projects.

HOW TO USE THIS GUIDE

The sections marked by (#) in the Table of Contents contain specific recommendations and guidelines for effective software management. This material should be utilized to develop appropriate administrative orders or management procedures to implement the recommended practices in all software projects. Selected references for further study of the more important topics are given at the end of this guide. Additional sections of this guide contain information that would benefit a senior executive in understanding the problems and importance of software management.

WHY SOFTWARE MANAGEMENT IS IMPORTANT

*Software is a very expensive resource for government.

The procurement and development of computer software is "big business" for many organizations, including the government. Estimates of Federal government spending range as high as as $5 billion annually on software, and the accumulated investment in current Federal software probably

exceeds $25 billion dollars. Software costs greatly exceed equipment costs over the useful life of computer services, and the initial development costs for government software systems are growing to enormous levels. For instance, cost estimates for the newly proposed Tax Administration System of the Internal Revenue Service are well over $500 million dollars, and the Air Force has estimated its software costs for command and control systems in the 1980's will be several billion dollars.

*The government software inventory is large and complex.

Among the more than 8,000 government computer installations, many have 500 or more programs for specific government tasks, in addition to others acquired from the computer manufacturer for general support of computer operation. Government computer applications, which include sophisticated military systems as well as public service systems handling millions of accounts, are typical of the most challenging uses of computer technology within the state-of-the-art.

*Software flaws and mismanagement can be ruinous to government projects and public services, and even to national goals.

Government and business are completely dependent on computer services today. Manual methods are no longer considered because human means cannot be organized to handle the volume of data and short deadlines that are needed in widespread public services. Social and economic activity is geared to the performance of computer systems. Examples are many, as in airline reservations systems or the Social Security Administration systems that distribute over $50 billion annually to about 25 million individuals. These benefits are possible only with effective management of the computer software. In recent years, recurring cases of major software failures have demonstrated the serious consequences of software mismanagement to legislators, journalists, and the public. One dramatic case occurred in 1975, when a government agency made more than $500 million in overpayments to individuals as a result of erroneous computer programs.

*No technology or standard practice now exists that will surely prevent faulty design, logical errors, cost overrun, or late delivery for any software project.

The preparation and maintenance of computer software is a laborious, human task that can require the coordinated effort of dozens or even hundreds of programmers. The

possibilities for program errors, poor design, or poor decision-making may be higher than in any other technical field, because of extreme technical complexity and the lack of standards and common practices in this new technology. The speed and volume of data handling by the computer creates disastrous consequences from errors in programs and accentuates the difficulties caused by only minor design flaws. There is no practical method known that will guarantee software as free from errors or design flaws. Although programming productivity has been improving, the advances have not kept pace with the tremendous growth in computer applications and the dramatic gains in performance versus cost for computer hardware over the past twenty years. There are very few standard computer programs for common applications. There are no standards covering common work practices and technical goals, such as the measurement of software reliability. Thus, every software project is a unique effort. Success or failure is determined more by the individual technical skills and management capability of the personnel involved, than by any other factor.

WHY EVERY AGENCY NEEDS SOFTWARE MANAGEMENT

*Software management must be the responsibility of the computer user organization.

Over 70% of all software is developed by the users of computers, rather than the manufacturers of computer equipment or the producers of commercial software. Although the computer industry is working to improve software management for its products, industry cannot be expected to provide a comprehensive solution, nor to cover the unique circumstances of each computer user.

*Software design and programming is a complex technical activity that must be directed effectively.

Although many people nowadays may understand the rudiments of programming from college or even high school courses, the simple homework exercises in such courses do not represent the "real world" of professional practice. The design and production of numerous programs that must work together for a major service, such as the payroll of a large business, is not routine effort that can go unsupervised or be done by junior personnel. The funds and personnel resources involved may be a major investment for the organization, and the technical choices may drastically

affect non-computer work processes in management, clerical, and client relations areas. Technically complex programs may demand advanced professional skill which is very scarce and must be carefully directed at the critical technical problems. Management and technical control by a professional software designer is essential for resolving complex design issues and for directing programmers to produce a specified product within cost and schedule targets.

*Software management establishes accountability for project decisions and objectives, and provides upper management with visible measures of progress toward desired goals.

Software projects usually are initiated by managers in such areas as payroll, engineering, plant and supply, etc. Although they may control the project budget and schedule, and have the best information on potential benefits and requirements, they are often unprepared to conduct a design project or to judge the results provided by an outside technical group. Thus, software designers sometimes work independently using their own judgement with little pressure to reliably estimate their progress and the remaining work or costs. Software management includes checkpoints to assure that important objectives and issues are carefully evaluated by the concerned parties, especially in regard to cost, quality, and schedule, and that users and designers take their appropriate responsibility for the decisions made.

*Software management provides the technical controls and resource management to produce high quality software within acceptable funding and time limits.

Software management emphasizes technical criteria and working procedures to resolve the technical issues that are critical to the success or failure of a project. Major questions commonly arise concerning the feasibility or performance of a desired system, the attainable quality of the system considering the resources available to produce it, and the methodology for design and testing in order to achieve the most effective software product. Software management delineates quality controls, standards, planning factors, and experience data which help designers and programmers to organize and direct their efforts efficiently in solving such problems within available cost and time limits.

*Software management controls costs after software operation begins.

The useful life of software for a significant service typically is over eight years. Experience has shown that continual changes are necessary in software, to correct previously unknown errors, to improve useability or performance, or to include new functions required because of legal, administrative, or technical changes. As much as 70% of software cost occurs in this redesign and maintenance, after software has been delivered and put into use. It is important to recognize that maintenance is no less challenging than development, for change to programs involves new design work and should be done according to planned cost and time goals. Software management procedures, consistently applied throughout the software life, will make best use of limited funds by stressing close management of effort toward the most needed software capabilities.

BASIC TASKS OF SOFTWARE MANAGEMENT

*Software management consists of all the technical and management activities, decisions, and controls that are directly required to purchase, produce, or maintain software throughout the useful life of a computer system or service.

Software management is primarily concerned with computer programs, data files, and documentation, rather than the computer equipment configuration that executes the programs. Even so, software designers must have a say in hardware decisions if they are to meet the goals for a computer service. The term "system" will be used here for the combination of hardware and software that provides a computer service. Since software has little purpose without hardware, the terms "system" and "software" may be used interchangeably without causing confusion.

The basic activities and decisions are best introduced in relation to the commonly recognized phases of a system's life cycle, shown in Figure 1. The INITIATION phase involves the assessment of an existing, inadequate system, in order to determine the feasibility of replacing it with an improved system. The four phases, DEFINITION through TESTING, are the development period, where a new or improved software product is being designed and implemented. The OPERATION phase starts with the operation of a new system by its intended users, and includes the subsequent maintenance and minor redesigns that user experience may dictate. Eventually, maintenance no longer suffices for needed

improvements, and another development cycle is begun.

Specific guidelines for each phase will be given after the following summary of the tasks included. Figure 2 helps to understand each phase by summarizing the documents produced that are needed for quality control.

*In the INITIATION phase, software management assures that the technical designers understand the purpose and scope of the envisioned system, and adequately perceive its required functions.

The INITIATION phase begins when management recognizes that a need exists for a new or improved computer service, and that a study should be made to formulate a software solution. During this phase, software planners or system analysts identify the intended users and investigate their work processes. The planners describe the services needed from the envisioned software, and then conceive and outline alternative software solutions. A close working relationship with the intended users is essential. The alternative approaches should be described in basic terms that users can understand. User concurrence with the stated requirements and the recommended software approach is mandatory.

*The INITIATION phase assures that the recommended software approach is technically and economically feasible.

An estimate must be made of the costs and benefits for each of the potential software solutions. This will lead to a recommended solution and provide data regarding feasibility. Comparable existing systems are investigated to derive cost data and to confirm that proposed solutions are technically realizable. Any pitfalls, failures, or questionable design areas in comparable projects should be covered thoroughly. Comparisons of the proposed project to past failures or problem areas would assure that they can be avoided or resolved. General specifications must be written to show clearly the scope and character of the recommended solution. The INITIATION phase concludes with a report delineating the recommended software and the supporting analysis. Management may then approve the concept and budget the recommended funds and personnel for acquisition and operation of the planned system.

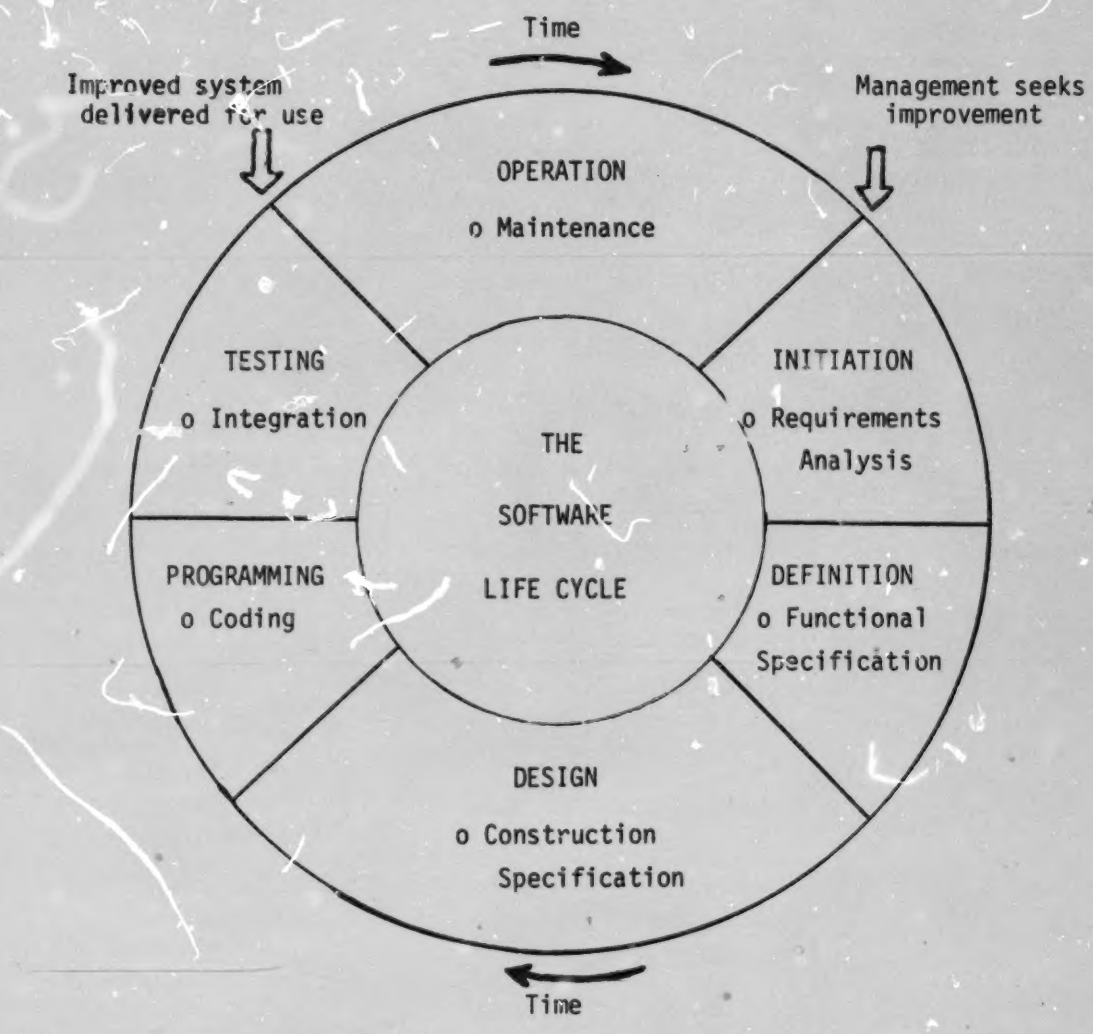
*The DEFINITION phase defines the functional and performance requirements, in order to confirm that the software, when built, would meet its objectives.

The DEFINITION phase begins when management commits resources to the design or purchase of the proposed software. In this phase, detailed specifications are created for those externally apparent functions and design characteristics that are crucial to the purpose and operation of the software in the intended environment. Input and output data, processing operations, and performance goals are defined. The Functional Requirements define what the software must do and the general aspects of how it will be constructed, rather than the details. The DEFINITION phase concludes with a specification that is sufficiently precise for outside contracting, if desired. Equally important, the specification allows the intended users to confirm that the proposed system will meet the need and will have acceptable qualities of useability, generality, etc.

*The DEFINITION phase provides a project plan for managing the acquisition and installation of the system.

A thorough plan for producing or purchasing the software is prepared as part of the DEFINITION phase. Projections are made of the detailed costs. A delivery schedule is laid out, including intermediate milestones and technical reviews for evaluating progress and insuring that the project will meet its goals. Working methods are defined for quality control of intermediate and final products. The plan should extend to the installation and initial operation of the system, and should include user training, document preparation and review, acceptance testing, and the continuing obligations of any contractors, vendors, or support organizations. The Project Plan is extremely important because it specifies how the effort will be managed, how problems will be resolved, and how the quality of the final system will be assured.

Figure 1. The Life Cycle of a Software Product



o Major technical activity

Figure 2. Chart of Basic Quality Control Documents

Life Cycle Phase	Documents	General Content
Initiation	Feasibility Study Report	Requirements analysis, definition and evaluation of alternative solutions, recommended software concept.
Definition	Functional Requirements	External specification of software functions and operation, including design constraints.
	Quality and Performance Requirements	Desired design attributes and quantitative performance parameters for key processing operations.
	Project Plan	Cost and work breakdown, detailed schedule including quality assurance milestones, and definition of primary management and quality control methods.
Design	Design/Coding Standards	Detailed design attributes and coding conventions for quality.
	Design Specifications	Software architecture, module definitions, interface specifications, and programming requirements.
	Test Plan	Testing criteria, testing schedule and work breakdown, standard test case specifications.
Programming	Review Reports	Identified discrepancies and recommendations from team reviews.
	Unit Test Reports	Results of unit test for each module.
	Inspection Reports	Results of inspection sessions for each subsystem.
	Software Manuals	Usage, operation and maintenance descriptions.
	Program Listings	Source Language statements for each tested module.
Testing	Test/Fault Report	Results of scheduled tests.
Operation	Fault Report	Symptoms, circumstances of detected or suspected failure.
	Specification	Functional and design specifications for maintenance and improvement work.
	Maintenance Plan	Work breakdown, cost estimate, and schedule for proposed rework.

*The DESIGN phase produces a thorough design specification for guiding a timely and problem-free implementation of the proposed software.

The DESIGN phase begins when the Functional Requirements have been validated by intended users, and implementation of the system is approved. The DESIGN phase results in detailed specifications of the internal construction of the software, for use by programmers who will implement the design. The DESIGN phase involves definition of the internal architecture of the software, performance trade-off analyses of algorithms, specification of individual programs or modules, specification of interactions between components of software, etc. The Design Specifications are important as a means for resource management and quality control during PROGRAMMING.

*In the PROGRAMMING phase, software management assures that appropriate technical choices are made, and that high quality workmanship is applied in producing code meeting the Design Specifications.

The PROGRAMMING phase often is merged with DESIGN and so may have no distinguishable starting point. PROGRAMMING involves final technical choices for internal program design and the creation of programming language statements or "code" that is executable on the computer and that implements the design. Software management here is the day-to-day supervision of programmers, and guiding and reviewing their technical results. PROGRAMMING effort includes the testing by each programmer of his individual programs and the preparation of pertinent management reports and software documents. Test reports, inspection reports, and program documentation act as tangible quality controls over PROGRAMMING accomplishments.

*In the TESTING phase, software management assures that no significant errors or design discrepancies will impede usage of the system in the intended operational environment.

The TESTING phase is concerned with verifying that the software product meets the functional specifications and contains no significant discrepancies or logical errors to impede its use. TESTING proceeds according to a Test Plan prepared in the DESIGN phase, and focuses on the integration of individual programmer results in an overall working system. Noted discrepancies and errors are documented for corrective action by the programming team. Gross shortcomings in the original design or functional specifications may be exposed; this should lead to a major project review in order to plan and organize additional

definition and design work.

*In OPERATION of a delivered system, software management assures that maintenance is as well managed and controlled as original development.

The OPERATION phase follows the delivery and installation of the software in the field, and involves periodic redesign and improvement. Software management procedures that are recommended during development can be streamlined to provide similar close control of the reduced effort committed to corrective design work. Thorough specifications, quality workmanship, effective testing, and management review remain highly important in directing technical effort to meet cost and schedule targets.

*During the development period, the earlier activities may be continued or repeated in order to remedy design defects and improve quality of the final system.

Although the successive phases defined here should be followed whenever practical, some flexibility is advisable, especially when a project involves unusual complexity or innovation. For example, it may be important to carry only a part of the system through to DESIGN and PROGRAMMING, in order to confirm feasibility before beginning DEFINITION for the entire system. Or, with limited effort available, development of some supporting capabilities or auxiliary programs may be deferred until the mainstream processing is successfully completed. So, all parts of the software need not be at the same stage of development, and the life cycle orientation may be viewed with some flexibility.

THE QUALITIES OF SUPERIOR SOFTWARE

These general properties are commonly accepted as characteristic of high quality software. The techniques of software management may be used to control quality at acceptably high levels or to maximize quality insofar as practical with the resources and time available for a project.

CORRECTNESS--Programs perform exactly and correctly all the functions expected from the specifications, if available, or else from the documentation. This means that incorrect documentation is as serious as an incorrect program. Correctness is an ideal quality, that is rarely

determinable, so a more practical quality is RELIABILITY.

RELIABILITY--Programs perform without significant detectable errors all the functions expected from the specifications or the documentation. High reliability indicates that programs are relatively trouble free in performing what they are claimed to do. But an equally important question is whether the functions and performance are adequate and suitable to a needed purpose. The latter quality is called VALIDITY.

VALIDITY--Programs provide the performance, all functions, and appropriate interfaces to other software components, that are sufficient for beneficial application in the intended user environment. This means the software, without additional programming or manual intervention, has the capabilities that reasonably would be expected for its purpose. Validity is a quality of specifications as well as computer programs. Examples of an invalid program would be an interactive editor that had no online function for retrieving stored text for inspection, or a FORTRAN language compiler that had no DO loop implementation. Validity involves judgement of user requirements, and may change if the intended application or purpose is altered. Because poor reliability may render a needed function useless, reliability is necessary to validity.

RESILIENCE (or ROBUSTNESS)--Programs continue to perform in a reasonable way despite violations of the assumed input and usage conventions. Input of unacceptable data, or an inconsistent command, should never cause a result that is astonishing and detrimental to the user--such as the deletion of any valid results obtained previously. Programs should include routine checks and recovery possibilities that are "forgiving" of common user and data errors. Resilience is related to the broader quality of USEABILITY.

USEABILITY--Programs have functions and usage techniques that are natural and convenient for people, showing good consideration of human factors and limitations. For example, the programs have few arbitrary codes for data in input or output, have consistent conventions in different operating modes, and provide thorough diagnostic messages for errors or violations of use.

CLARITY--The functions and operation of the programs are easily understood from the user manual, and the program design and structure are readily apparent from the listing of program statements. This means that documentation must be well written, but also that the program is carefully

designed, with meaningful choices of variable names, use of known algorithms, frequent and effective comments in the program to describe its operation, and a modular structure that isolates separate functions for examination.

MAINTAINABILITY--Programs are well documented by manuals and internal comments, and so well structured that another programmer could easily repair defects or make minor improvements. Clarity is essential for maintainability. But also implied are a wide variety of good design attributes, such as program functions that help to diagnose potential problems, e. g., periodic reports of status or control totals; or, general techniques that can be readily adapted for change, e. g., the isolation of constants, report titles, and other static data as named variables.

MODIFIABILITY--Program functions that might require major change are well documented and isolated in distinct modules. Maintainability is essential to modifiability. But modifiability means that a concerted effort was made to anticipate major changes, and to plan the software design so that they could be made easily.

GENERALITY--Programs perform their functions over a wide range of input values and usage modes. Programs are not limited to special cases or ranges of values, when the functions are commonly or reasonably extendable to a more general case.

PORTABILITY--Programs are easily installed on another computer or under another operating system program. Standard programming language is used, and hardware or other software dependent features are isolated for easy change.

TESTABILITY--Programs are simply structured and use general algorithms, to facilitate step-by-step testing of all capabilities.

EFFICIENCY/ECONOMY--Programs have high performance algorithms and conservatively use computer resources, such as main storage, so that the cost of program operation is low.

ORGANIZING FOR SOFTWARE MANAGEMENT

---

*A project team is the basic organization for software management.

The team approach is effective because a mix of needed skills can be quickly assembled from an organization's available talent, and the most talented designers can be consistently utilized on the toughest problems. Team practices provide better oversight and counseling through peer reviews, better dissemination of technical and status information about software modules, and better back-up of personnel. The project team approach will be an unusual practice where computer personnel have been organized according to specialties such as "operations", "system analysis" and "systems programming". Nevertheless, the advantages gained by forming an appropriately skilled team, dedicated to one product objective, warrant a departure from the traditional organization. A project team may be formed temporarily to carry out software development, with further effort after installation handled in the normal organizational framework. However, the advantages of team work apply to maintenance effort in the OPERATION phase as well as to the development phases. So a project team may be advisable when major maintenance or improvement activities are done.

*For projects involving up to about 10 personnel, one individual should be responsible for the entire system design and the project management.

This concept has been called the "Chief Programmer" approach, for the individual must be an exceptional programmer as well as a very competent manager. Control through one individual, especially from DEFINITION through TESTING, offers a better possibility of a consistent, high quality product, than would cooperative work by several individuals or smaller teams. The team chief personally defines and designs the entire system, and produces many of the critical modules. The chief exercises complete authority over the technical results and team production. He or she represents the project with users or customers and with higher management, and decides the project commitments made to them. The chief controls personnel assignments and schedules, and has supervisory responsibility for all team members. The team chief usually should have a minimum of five years of diversified, intensive experience in system analysis and software design, and should have demonstrated the personal qualities, such as judgement and maturity, that are needed for effectively supervising other specialists and for dealing with clients and superior officials.

*The project team should include an assistant chief, specialists for key problem areas, and supporting personnel.

The assistant team chief substitutes for the chief whenever necessary, and serves as his primary technical advisor. The assistant may personally define, design, and program some parts of the system. When needed, particular specialists become team members to handle difficult design problems requiring their unusual knowledge and experience. In PROGRAMMING especially, a programming language specialist should be available to answer questions about the effect and performance of various instruction sequences, and to suggest better ways of programming certain algorithms. Specialists on program production tools, testing, and data management techniques are advisable also. The team secretary assists in clerical tasks aimed at improving the team productivity, such as the preparation of computer runs, purchase of computer supplies, handling of project records and software documents. Additional support is advisable also in documentation and technical writing, and in administrative support of the team on personnel matters, training, management information support, etc. Figure 3 depicts a representative team structure.

*Team techniques are appropriate also for very large projects.

When more than about ten people are needed to do a project, one individual cannot effectively supervise all of them, so the design and programming must be distributed among several teams. Nevertheless, one manager should be given authority over the entire effort, and should make the fundamental decisions on work breakdown, resource allocations, and software design. This project manager and his subordinate programming managers could be viewed as comprising a team to carry out system design and project management. Their working procedures should be similar to those within a subordinate programming team; for instance, conducting periodic design conferences where each team chief presents the designs and other results of his group. The project manager should exercise considerable technical and design leadership, and major aspects of overall system design would be decided by his team. Support arrangements similar to a programming team are advisable, as suggested in Figure 4.

*Team techniques are applicable in managing contracted developments.

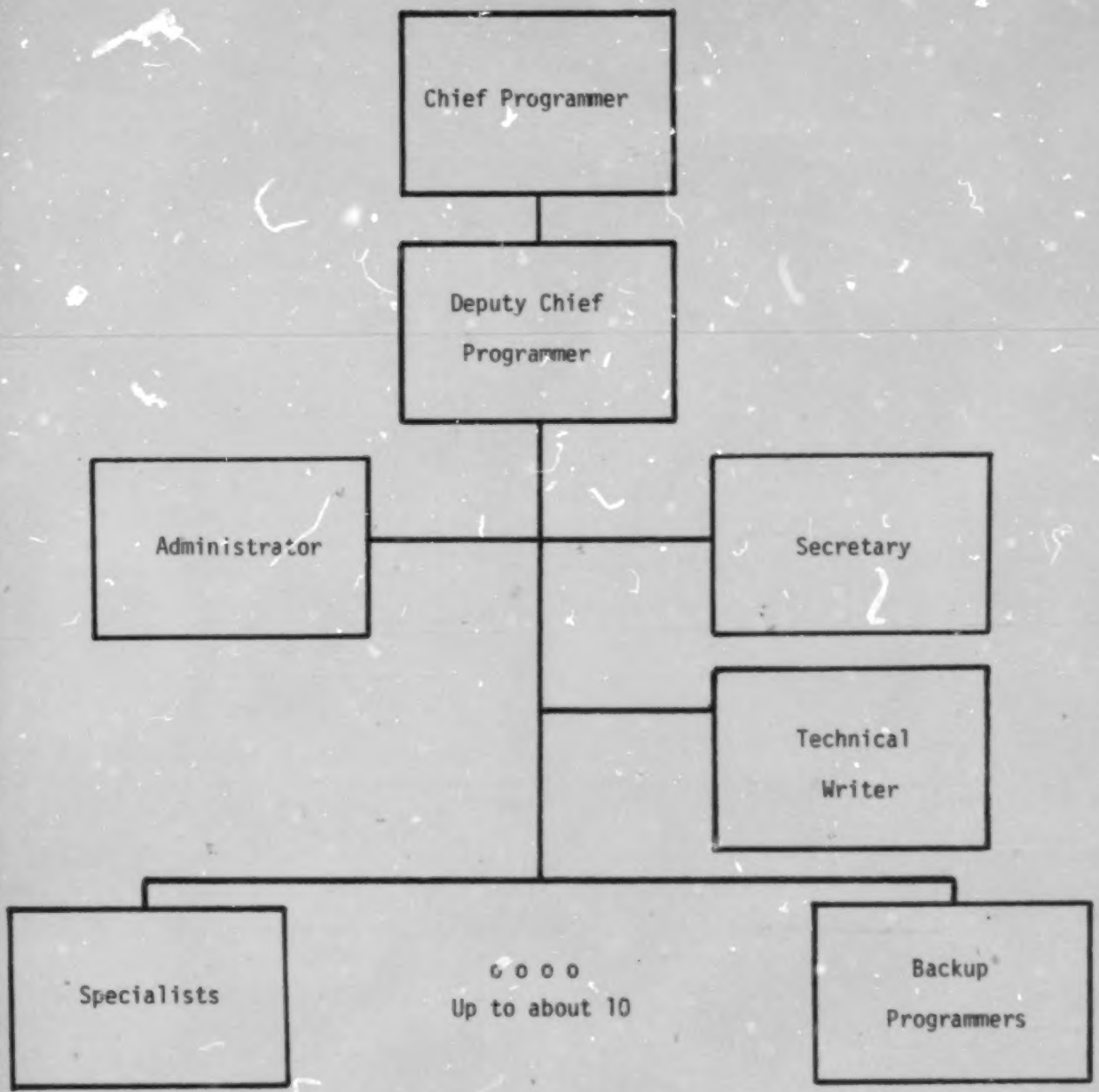
A team approach is advisable also when the software development will be accomplished principally by private contractors. The project manager or team chief will serve then as the contracting technical monitor. The contracting team, rather than designing and producing programs, will

prepare the technical statements of work, and other contractual requirements such as the quality and performance specifications, and the project plan in regard to mandatory schedules, deliverable products, and quality controls. Technical specialists in particular areas such as testing and performance evaluation are also necessary on the contracting team, and special administrative support is also advisable to improve the quality and timely completion of contract requirements, technical reviews, acceptance testing, etc., as the contractors proceeds with the development.

*Agency-wide direction and evaluation of software policy and quality control is needed in addition to individual team management.

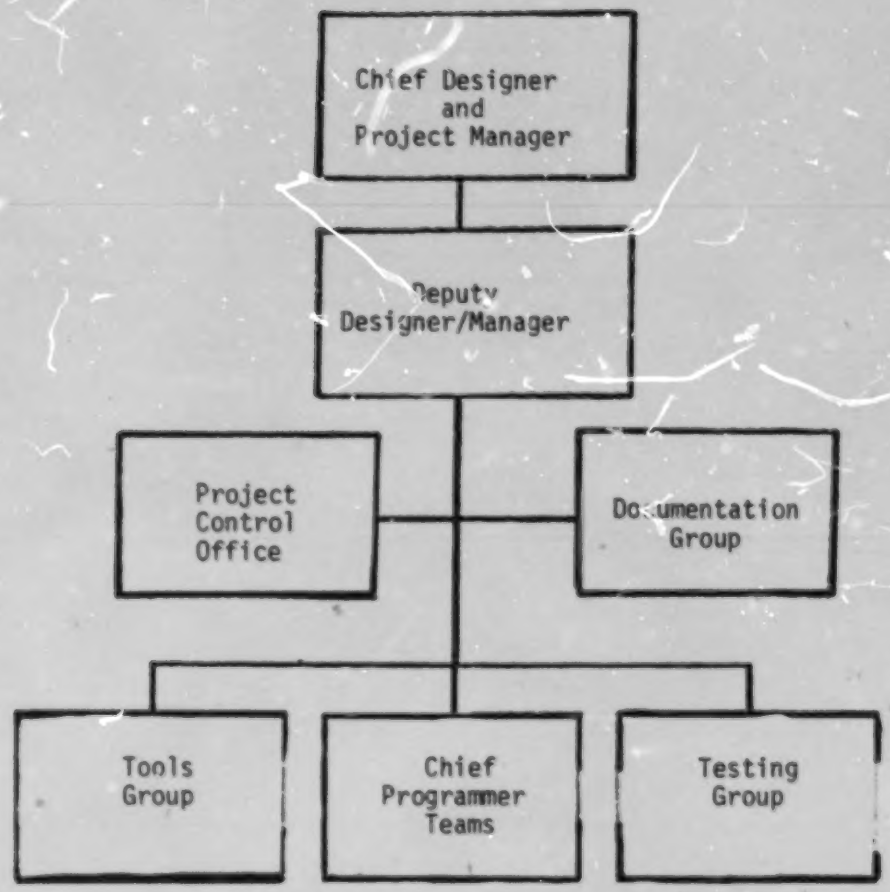
An upper level of software management authority is advisable to provide consistent direction and support for individual teams with regard to organizational policies, standards, and objectives in software. Such an individual or office can contribute significantly in assessing software technology for project needs, and in reviewing project accomplishments with a view to standardizing team procedures, technical aids and software production tools, and even specific software components that have wide utility for the organization.

Figure 3. A Representative Team Organization



- o Language expert
- o Testing specialist
- o Tools specialist

Figure 4. A Representative Organization for a Large Project



MINIMIZING EARLY PROBLEMS

*Software problems originate most frequently in the analysis of requirements and the design of the computer software.

Nearly 50% of software flaws and errors exposed in final testing or initial use of a new system are traceable to shortcomings and mistakes in the early stages of a project, i. e. from INITIATION through DESIGN. Most authorities suggest that added effort in the early stages is the best way to improve the quality of software. The effort up to the start of PROGRAMMING should comprise about 40% of the total development effort. A greater allocation is advisable, provided working methods and review techniques are geared to strong improvements in the validity and detail of the Design Specifications.

*The primary quality controls for the INITIATION and DEFINITION phases are system requirements documents and the technical reviews made of them by the parties concerned.

Quality control in these early stages is necessarily judgemental, but will be aided considerably by standardized documentation practice. The National Bureau of Standards has prepared guidelines, FIPS PUB 38, that indicate the content for functional requirements, project plans, and other software documents. Additional guidelines and standards will be issued in the near future. The use of computer-aided text editing and document formatting will be especially helpful in rapidly accommodating numerous small changes in specifications during repeated and lengthy reviews.

*Start a Project Record with the INITIATION phase and continue it over the system life.

The Project Record should be an open record of the major decisions and of the pertinent documents and specifications. It should be official, to clearly indicate firm commitments, yet openly available, so that inadequacies can be detected as early as possible and so that current goals and constraints are apparent to all. This record is distinct from a technical notebook, which also is often effective as an informal medium for exchanging tentative proposals among designers and programmers.

*In the INITIATION phase, identify all user tasks to be supported by the software and the important attributes of those tasks.

The documentation of the INITIATION phase should characterize the working environment in which the proposed software will operate. This would include a description of user tasks to be supported, the present methods and computer aids for carrying out these tasks, and the present limitations that the proposed system would overcome. The description should be oriented for the users' review, so that they may identify omissions or shortcomings of the proposed software solution.

*Maintain a continuing interaction with the users and their managers during the development period.

The potential users or their managers are authorities on the working environment in which the proposed system must function. They can provide vital information on the acceptable performance and design goals for the system, such as input data characteristics, tolerable response time, etc. Their strong support and involvement increases the likelihood of successful planning and design, and helps in obtaining needed requirements data and evaluations of the proposed system. A highly recommended approach is to have selected users serve as members of the project team, not necessarily to perform design work, but to help to decide and to document requirements. Project milestones should include scheduled review sessions as the requirements and alternative solutions are evolved, to obtain user comment and concurrence.

*Straightfoward automation of existing manual systems may not yield the greatest benefits from computers.

The computer's capability to do complex processing of large volumes of data at high speed generally means that computers can provide better analytical assistance to human decision-makers than could any number of clerical people. The way in which data records are organized for manual handling also may not be the best to use in computer storage and retrieval. The INITIATION, DEFINITION and DESIGN phases should include research toward new algorithms and data handling procedures that allow the computer to serve as more than a high-speed clerk and file cabinet. For example, there are mathematical procedures that can greatly improve inventory controls, scheduling for scarce resources, and plans for performance of complex tasks. Also, there are well proven computer techniques that provide rapid retrieval of the very data needed for a human task, rather than requiring people to manually hunt through long computer print-outs of all related data. Improved algorithms, carefully chosen for each application, can increase human and computer productivity by many times.

*Radical advances in automation carry exceptional risk--experiment with new approaches through prototype software, and be prepared to throw away one or more systems before confirming the design.

Too often, the cause for software management failure is overambition or executive zeal to bring the benefits of automation to a pedestrian manual operation in one great leap forward. Never undertake to produce in a single project step a system for which there is no technological precedent. Keep the scope of systems within the bounds of your experience with successful projects--advance your system capability in modest increments.

*Use consultants and visit other organizations to identify potential problems in building a system and to verify that system objectives are attainable.

Systems that have any possibility of challenging the state-of-the-art must be planned cautiously. Researchers, consultants, and organizations who have pioneered an innovative type of system can help define the major pitfalls and how they may be avoided.

*Systematically evaluate available software of a comparable type, whether commercial or other-user developed, before making a firm commitment to develop a new system.

The feature analysis technique is very useful for evaluating available software, and comparing proposed system requirements against the current state-of- the-art. Feature analysis proceeds by decomposing the requirements into individual functions and attributes, and then developing short descriptions of what the available software provides for each feature. The descriptions should be organized in a table for easy comparison. Available software, if adequate to the need, can usually be purchased for much less than the cost of producing similar programs. Even if new development is required, existing software may serve for some components and reduce overall system cost.

*Formulate and evaluate alternative solutions that differ in the risk and cost of development.

Prudent management requires a look at other choices before deciding upon one system to be bought or built. Since software cost and risk are now so important, they naturally are key considerations in conceiving of competing approaches. The possibility of meeting the need through commercial computing services or other sources of available software certainly should be presented as one alternative,

if feasible. No definite guidelines can be offered on how to formulate alternatives. Clearly, thorough knowledge of the state-of-the-art is essential and will indicate candidate solutions that are more or less innovative.

*Estimate the future system costs by several techniques and adopt a conservative projection.

There is no well formulated cost estimation method that is highly favored and widely used. Various published data exist on programmer production rates, but different variables are involved and some important factors are omitted in one case or another. BROOKS (see References) gives a concise summary of published data. YOURDON and WALSTON also have data showing the gains possible with recent innovations such as structured programming. Experience and judgement, however, are the most accepted means to reach an overall recommendation on the time and funding needed. Published data should certainly be used as a guide, noting carefully the pertinent factors included. Examine costs of comparable systems and advertised costs of commercial packages for added evidence. Do a detailed cost breakdown of the system by major components and by major technical activities or work processes. Include operation and development costs, to have a full life cycle investment comparison of alternative systems. Above all, recognize the uncertainty of any early estimate of cost, and choose a conservatively high, yet reasonable figure. Reevaluate the cost estimate periodically as the development proceeds and better definition of the software evolves.

*Have the intended users review and approve the requirements and recommended solution defined in the INITIATION phase.

User concurrence, along with any evaluations and recommendations from the user review, should be in the record concluding the INITIATION phase. This could be as simple as initials in a project notebook for a very small project, or may require an official report and approval letter for a large project. To reach their assessment, users should be given a cost and benefit analysis of alternative systems, in addition to the general specification of each.

*Resist generalizing the system capability to accommodate requirements that cannot be resolved before DESIGN begins.

If an imposed schedule will require starting DESIGN before all requirements are defined, a judgement must be made of whether the uncertainty allows useful effort to proceed. If it does, then the unknown requirements must be

anticipated and the most likely possibilities defined. The impact of these alternatives on the system design must be determined, and if great enough, it may be necessary to proceed with definition and design of several systems until the unknowns are eliminated. Be cautious about generalizing the required system capability to handle arbitrary, unknown demands. Generalized systems often involve great risk in development, and may have poorer performance and efficiency than more constrained and specialized designs. Be forceful in investigating potential needs, in order to resolve requirements questions and uncertainties as early as possible.

*Use engineering analysis, modeling or simulation, and experience data to establish the performance needed in critical software operations.

Software designers often assume that necessary performance can be attained by "tuning" programs once they are running. On the other hand, early decisions, for instance on the hardware, the programming language, or the principal algorithms to be used, could be so poor as to eliminate any hope of reaching a needed level of performance. Engineering analysis, coupled with measurements from comparable existing software, should be used to investigate the expected system performance as a function of the attainable performance for basic processing steps. Performance goals should be specified for the critical operations, to support feasibility judgements and to serve as validation criteria for the DESIGN and PROGRAMMING phases. Early decisions that constrain the design or implementation approach for the software should show clear evidence that the necessary performance would be achievable.

*Include with the Functional Requirements the performance and quality objectives that the delivered software must meet.

The specification of quality criteria should begin in the DEFINITION phase in order to have a basis for judging the subsequent design. Use the definitions given above as a starting point, to be expanded and specialized for a given project. For example, identify the system functions that are most subject to future modifications and the nature of the required modifiability. As another example, describe such useability characteristics as the processing points where recovery procedures are required and the acceptable form of recovery. Include the quantitative performance goals for principal algorithms and processing steps. In the Project Plan, define the technical reviews and the evaluation

procedures that will be used to assure that the quality and performance objectives are met throughout the project.

*Develop the Functional Requirements in a structured format, to expedite review and validation.

A free-form narrative approach to documenting requirements will make it difficult, except for very simple programs, to isolate individual functions and performance figures for later confirmation of design. A better approach is first to decompose overall requirements into major functional groups, such as data entry or report generation, or into processing modes, such as online query or transaction auditing. Then describe individual requirements under each group with a separate statement of a function that must be provided or a performance or design attribute that must be achieved. Each requirement should include criteria for confirming its achievement, the testing or evaluation methods involved in confirmation, and any comments or discussion that amplify the requirement statement or show its origin.

*Do not conclude the DEFINITION phase without having an approved specification that incorporates changes and refinements emerging from user reviews.

Never proceed into detailed design when ambiguity or known errors exist in the requirements specifications. Basic disagreements are more likely to lead to fundamental redesign than to minor and easily rendered change.

*Formulate concrete, measurable milestones of progress for the design, programming, testing, installation, and initial operation of the system.

It is impossible to exercise control and take corrective action without concrete evidence of where problems lie. This means that progressive milestones must be conceived covering the life of a project. They must be events and accomplishments that are directly observable by the software manager, such as the receipt of a specified document, the successful operation of a given program, or the conclusion of a scheduled review meeting. They must be sufficiently frequent over time to serve as progress indicators or trouble alerts if not completed. The dependence of later milestones on earlier events should be revealed by drawing up a graph or network relating all milestones. Progress during DESIGN and PROGRAMMING should be measured by required events, such as reviews or specifications, and not by programmer estimates or percent of code completed. The accompanying chart describes a

minimum recommended set of milestones over a system's life cycle.

*Establish controls and milestones that will identify and correct errors or omissions as early as possible.

The cost of correcting an error or adding capability to a system increases significantly as a project proceeds. By the time system testing begins, errors are over ten times more costly to fix than if they had been recognized in design. Additional time given to improving specifications and to thoroughly critizing them is worthwhile in reducing later project costs.

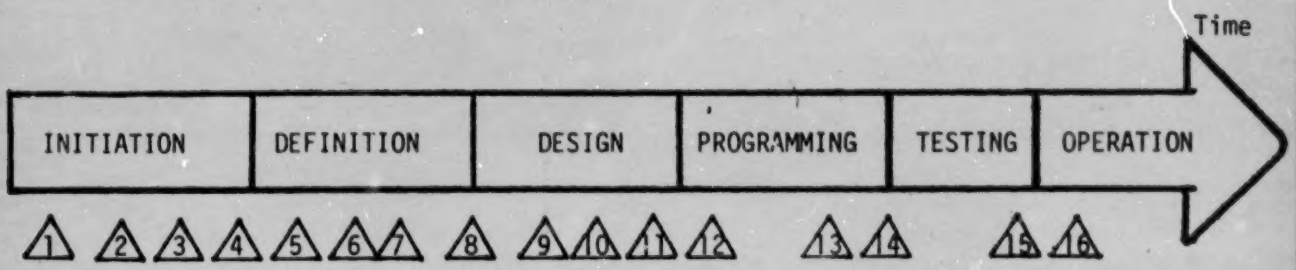
*Plan at least 10% of total project effort for management requirements and technical reviews.

When the needed effort has been underestimated, which too often is the case, careful reviews and management analysis may be thrown aside to put all effort into design and programming. Obviously this will work to the detriment of quality objectives, and probably will result in even more time required for testing and rework later. Make sure that project estimates include the time and effort for detailed review of work as it progresses, and for continuing review of project schedules and resource expenditures. Insist on using this time as intended, so that project status and problem sources can always be accurately defined.

*Plan contracts for design and production only after the detailed Functional Requirements and Project Plan have been prepared and approved.

Contractors cannot be selected or effectively monitored without the quality assurance and validation criteria that are embodied in these documents. They also provide the first practical basis for decomposing the development effort into work packages that different contractors might undertake. Contract requirements particularly should call for standardized documentation and coordinated format among the Functional Requirements and the succeeding design specifications, to aid in tracing and correlating results as the effort proceeds.

Figure 5. Chart of Recommended Minimum Milestones



Description of Milestones

1. Initiate Project Record with plan for feasibility study.
2. Complete analysis of user tasks to be supported.
3. Complete definition of alternative software solutions.
4. Complete feasibility study report with recommended solution.
5. Obtain user concurrence with recommended solution; budget approval for project.
6. Complete Functional Requirements.
7. Complete Project Plan.
8. Complete user validation of Functional Requirements.
9. Complete software architecture and definition of major modules.
10. Complete Test Plan; and Design/Coding Standards.
11. Complete Design Specification of all component programs; tools definition.
12. Complete validation of Design Specification.
13. Complete unit testing; accept all modules for source code control.
14. Initiate fault reporting procedures.
15. Complete integration testing.
16. Complete delivery and shakedown.

GUIDELINES FOR DESIGN AND PROGRAMMING

*The second major source of problems lies in the quality of programs as they are produced.

About 40% of software errors are traceable to misinterpretations of specifications, errors in designing to them, or mistakes in writing program statements. Recent research to improve software management has concentrated on these problems. A variety of new techniques, such as structured programming and automated quality control, are beginning to come into widespread practice, but much more must be done. Misinterpretation of specifications would be reduced by more formal or mathematical specification methods rather than exclusive use of narrative English. Analytical methods of design, replacing the subjective criteria used heretofore, would reduce the frequency of programming errors and costly maintenance. Software managers must keep current with such developing areas in software engineering research. Continuing training programs, special in-house workshops, and experimental projects are good avenues for stimulating the computer staff toward improving their working practices and technical knowledge.

*On beginning the DESIGN phase, prepare an explicit specification of the quality design and programming standards that must be observed in the DESIGN and PROGRAMMING phases.

The Functional Requirements should be accompanied by definitions of mandatory quality objectives, that serve as a starting point. Using these, a list of design attributes can be specified, extending in detail to conventions on the use of programming language features and constraints for individual program design. This expansion of quality specifications therefore will encompass the Coding Standards to be used in the project, and will provide a framework for starting design of individual modules. For example, the quality standards may define the extent of diagnostic messages to be provided, the types of data errors to be detected by validation algorithms, the manner of isolating particular processes subject to future modification, etc.

*Develop the Design Specifications as a hierarchy of processes, specifying the input and output of each process, and the data flow interconnections.

A hierarchy is an effective means to decompose functional requirements into successively more detailed operations. The approach is well matched to what is called "top-down" design for software, which proceeds by successive refinements of general functions, until arriving at the component modules or programs that must be implemented to build the system. Documentation for this functional design approach is rather simple, and it portrays the overall system operation in an easily followed graphical way. The References include books and reports on the technique.

*Complete the Design Specifications before beginning to code.

It is vital that the design be completely worked out before programming begins, so that programming effort is not wasted on erroneous design notions and so that programming, when started, has a complete guide to assure quality work.

*Decompose the system into program modules that are simple and easily understood, usually meaning about 100 program statements in size.

The advantages of small modules are many. Their functions are more clearly and simply describable. This helps direct the programmer toward a clear objective. Their brevity improves clarity and speeds review for validity and quality. Progress may be more accurately evaluated when the units of programming work are small and readily understood.

*Define general service modules to reduce duplication in program modules.

In designing the system into modules, keep track of replicated functions and define modules that provide such functions as services that all programs may use. These service modules usually are those that manipulate standard data structures in a system of programs, for example, entering items in tables, retrieving items, managing storage allocation, etc.

*Use standard high level programming languages.

The use of vendor-unique programming language, especially machine or assembly language, should be prevented, except for very small portions of code where assembly language may be essential to achieving the performance goals of the software. These exceptional cases should be strictly controlled by explicit approval. Keep a standard high level language version of assembly programs for documentation and maintenance assistance.

*Use structured programming techniques.

Structured programming is a concept that encompasses programming team management, design methods, and programming technology. Numerous books and articles are available, describing the design and programming techniques especially. Published results from evaluation projects confirm that increases in programmer productivity and reductions in program errors are achieved, often as factors of two or greater. Investment in the training and management orientation needed to practice these methods will be worthwhile.

*Identify or develop programming aids to increase team productivity, and issue guidelines to the team for using them.

Specific recommendations on programming tools are given later. Selection of tools and specifications for unique tools needed for the project should be done as early as possible in the DEFINITION and DESIGN phases.

*Conduct weekly reviews by the programming team of individuals' work.

This practice helps all team members to keep informed of the design and production as it proceeds, and may act also as a training opportunity. But primarily it focuses the combined talent of the team on the quality of the team output as a whole. Errors or inferior design can be more quickly exposed than with reviews by a single manager. The results of each review session should be documented as a guide for the individual programmer to improve or correct his work. In structured programming, these review sessions are called "structured walk-throughs", to suggest that each programmer leads the team through the design and operation of his module in the same pattern as it was developed.

*Design and program for quality before performance.

Before programmers make any effort to tune their modules for improved performance, all quality characteristics required should be realized. Necessary comments and program arrangements for clarity, maintainability, modifiability, generality, and all required functions should be completed and confirmed in reviews.

*For large projects, conduct inspections at successive stages of design and production, and make all necessary changes before work moves to the next stage.

In addition to team reviews of each programmer's progress, more formal inspections are recommended for large projects, to examine subsystems or closely related modules produced by different programming teams. These inspections should serve as progress milestones in the Project Plan. The inspection team may include experts outside of the teams, in addition to the team chiefs and the responsible programmers. The inspection team leader (or moderator) should not have been a participant in producing the subsystem under review and must have the authority to inform higher management of negative findings. A team size of about ten persons maximum is advisable, and separate reviews should be made when design is complete, when code and unit tests are complete, and when first integration tests are complete.

*Use iterative review techniques for validation.

Validation, also sometimes called verification, means confirmation that a system is reliable, and meets its specifications as well as the requirements of the user environment. Team reviews or inspections should always trace design specifications back to functional requirements, taking note of findings from past reviews. In this way, the logical thread that stems from the original specification is followed to the working software, and successively examined for continuity and possible shortcomings. Reviews should evaluate the suitability of the software for the intended use, and whether the necessary quality is being achieved.

*In the PROGRAMMING phase, the individual programmer performs "unit testing".

Unit testing determines that the individual programs work in isolation from others and meet the specifications given with the assignment. Unit testing must be performed according to standard criteria, rather than ad hoc methods contrived by each programmer. The programmer provides test reports for use in review and inspection tasks. Testing criteria are described in the guidelines for TESTING, which follow.

*Put program modules under source code control when unit testing is completed.

Completion of unit testing signifies internal project delivery of a module for integration with others in the overall software system. Source code control restricts further program changes to formally approved changes that have been decided from an overall system viewpoint as necessary to meet project goals. Source code control is

needed for best use of programming effort in the terminal period of the development schedule.

*Before the DESIGN phase is complete, a formal test plan for the system should be approved.

The Test Plan should be started in the DEFINITION phase, for the Project Plan must account for the effort required for the testing strategy to be used. Some details on the number of tests, test inputs, etc. cannot be specified until the design is nearly complete. The Test Plan describes the TESTING phase effort and the testing schedule, and defines the standard test set, i. e. the complete range of test cases that will be repeatedly used to assure reliability of the delivered software. The development of this standard test set is a significant effort that must be completed by the start of TESTING. The test cases must be provided by manual calculations or other independent means, so that the correct output expected is known beforehand. Other means may include program analyzers, simulations, tables of random numbers or function values, results of comparable systems, etc. The depth of the Test Plan is indicative of the reliability the user may expect if testing is successfully completed. Ideally, the Test Plan is developed by a technical group that is independent of the designers and programmers, and that can require tests that focus on perceived weak points in the design. Users should be represented in test planning, and the final plan should be jointly agreed between users and the chief designer.

GUIDELINES FOR TESTING

---

*Delivered software for large systems typically has errors at a rate of one in every 300 program statements.

When TESTING begins, the errors present may be three or more times higher than this figure. Testing is the process of executing programs with representative input data or conditions, for which the correct results are known, to determine whether incorrect results occur. Testing provides information that helps to isolate errors. Debugging is the

subsequent work of a programmer to explicitly identify errors and to correct them by changes to program instructions or data. Many large program systems never become error-free, i. e. "correct". Instead, they are maintained and managed in terms of the number of known errors that are allowed to exist--i.e. successive versions or releases are planned to reduce the known errors to some acceptable, but perhaps still high figure.

*Testing is the only practical way to detect program errors so that they may be eliminated.

Testing only determines if errors are present. It cannot verify that no errors exist in a program. Such confirmation is conceivable in only two ways. One is to exhaustively test the program on every possible input. Even for simple programs, this could take millions of years even if done automatically by a computer. The second conceivable way is to prove correctness as a logical and mathematical theorem about the program instructions. This approach is a current research topic in computer science, and proofs have been done manually for simple programs up to a few hundred instructions. Research work is now aimed at simplifying proof requirements through programming practices suited to this objective, and through computer theorem-proving software that would reduce the complex mathematical effort involved. No break-through is predicted in the near future that would make proof-of-correctness practical and economical for the average programmer to use on normally complex programs.

*Adopt specific criteria to govern unit testing of all programs.

Many different testing criteria are conceivable, but most have no direct relationship to possible errors and design flaws, and so provide no information to improve reliability. For example, one could require a specific number of test runs per program. But unless each test was appropriately distinct, no more benefit would be obtained than from running only one test. A random sample among most of a program's input values may be an effective criterion. But it is useful only if the valid input values are small in number, so that a significantly large number of random tests can be generated and run within the time and resources available. Other possible criteria are:

a) test every instruction at least once;

b) test every branch of the program;

c) test every logical path within the program with at most one iteration of every loop.

(A logical path is a sequence of instructions, some possibly repeated, from input to termination of a program.) The above three criteria are successively more stringent; they are preferable to many others because they are well directed at possible faults, and tend to be efficient in using test effort.

Recommended testing guidelines that are broadly applicable to software cannot be simply stated. The subject is technically complex and beyond the scope of this guide. The National Bureau of Standards is currently preparing a guideline that will include results from current research to develop testing theory and to lay down broadly effective principles. The most important consideration at this time is to avoid arbitrarily chosen tests by each programmer, that do not clearly exercise previously untested functions and logic.

*Schedule progressive tests to build up to a representative full system test.

Testing should begin with discrete tests of basic system functions that are needed in order to do more complex and realistic tests. Simple input, output, and control functions are examples. Keeping each test restricted to a given function allows the results to clearly identify problems encountered. Ultimately, the software should be exercised by composite test cases that represent full system operation and expected usage conditions. Composite tests should be conceived to stress the software performance. The limits on such factors as accuracy, acceptable volume of data, or number of concurrent users, should be verified if included in the specifications. The diagnostic features of the software should be exercised as well with test cases having invalid input. The Test Plan should encompass most, if not all, of the capabilities defined in the Functional Requirements.

*Use program analyzers to assure that all program functions have been exercised.

A program analyzer is a computer program that collects data about another program, including data about it as it operates. In particular, an analyzer can determine whether test inputs have caused all the program instructions to be used. This verifies that all capabilities have been exercised, and identifies extraneous code that may have no purpose and should be removed.

*Experience indicates that testing usually requires about 40% of available development time.

Careful, disciplined design should reduce the need for extensive testing, but more theoretical development and experience with improved techniques will be needed to assure this is always so. Certainly, better specifications would eliminate rework that sometimes is initiated during TESTING when it becomes clear that the design was incomplete or haphazardly contrived. But, it seems best to be conservative in project schedules as well as cost estimates, so allow a major portion of project time for testing and design rework.

*Use a fault reporting process to manage debugging and testing.

It is good practice to have a formal report system by which detected errors and discrepancies are recorded and fully described. These reports will help to confirm that all known errors are fixed before delivery. They also help to trace multiple instances of the same anomalous behavior, so that debugging assignments can be made for related problems and the debugging effort reduced.

*Conduct regression tests after program rework has been done.

The TESTING phase is primarily concerned with integration testing, i.e. tests to determine whether different program modules, produced rather independently, will work together correctly. When errors are found and eliminated by debugging, new undetected errors may be introduced. Regression testing means that previously successful tests are rerun to detect any introduced errors. A complete, successful run of the entire standard test set should be accomplished on the software before concluding TESTING and delivery to the customer.

*Supplement integration testing with system validation reviews.

Although specifications are expected to be of high caliber when PROGRAMMING begins, practical use of the resulting system during TESTING may reveal that original notions of desired operation were fallacious. Reviews of test sessions should consider the general suitability of the system as well as strict adherence to discrete functional specifications. Any discrepancies should be considered for rework, rather than delivering a system that in the judgement of users and designers will not be adequate for

the user's purpose.

*Testing should assure a high measure of reliability that would be expected in operation of computer programs.

Correctness is the desired quality for software, but also is an ideal condition that often is not economically attainable or confirmable for complex software. So, the term "reliability" is useful to denote the lack of detected errors with respect to some basis of observation. Observations could be made over a long time period of repeated use of the software, and reliability then defined as the ratio F/N, with F the number of detected faults, and N the number of runs or trials of the program.

Software reliability measurements will reflect how the software is being used, as well as the faults or errors it contains. If routine usage is limited to previously debugged functions, reliability will apparently be high. If usage changes, reliability may suddenly degrade seriously, as previously unexercised code is used and its errors revealed.

Reliability measurement is a recent subject of software engineering research. More results are needed before a minimum acceptable figure can be recommended. The most effective use of reliability data in project management also is not certain until more project experience is at hand.

*Plan a shakedown period after delivered software is installed.

A shakedown period may be dictated by hardware alone if a new computer system is involved, but this kind of trial use may also effectively supplement the previously performed integration testing of software. The shakedown scenario should be as close to the actual anticipated usage as costs and procedures permit, but do not depend upon the shakedown operation for any operational service. Use the fault reporting process and records of shakedown run time to measure and evaluate reliability. The shakedown period needed to attain stable and reliable operation may be several months, particularly if hardware installation is extended over this time and considerable user training is being done.

*Use the management techniques for development during the OPERATION phase as well.

The programming and design effort available during OPERATION is generally reduced compared with the development phases. However, the goals of software management do not change; indeed, maintenance costs may dominate the total life cycle cost of a software product. Maintenance work on software is still a design challenge, but one constrained by the program structure and instructions already in use. Unless technically controlled, maintenance may degrade reliability and other qualities that were initially present. Maintenance tends to disorganize and complicate the program structure, increasing the potential for errors and the cost of future changes. It should not be assumed that the existing software is a sufficient constraint to prevent unsupervised maintenance from becoming costly or haphazard. Maintenance should be conducted under a tight management regime that includes definition and scheduling of manageable work assignments, preparation and review of specifications, team review of design and code, standard tests, and documentation.

*Keep a failure and discrepancy reporting process to continually manage software reliability.

Failure reports are normal in the TESTING phase and the initial period of OPERATION, but they continue to be important as maintenance continues over the years. These reports are needed for software managers to accurately determine if maintenance work is being accomplished as assigned and if greater testing and rework effort is warranted. Be cautious about proposed improvements that are not strongly supported by users or indicated by problem reports.

*Evaluate software periodically and plan improvements to avoid obsolescence and to minimize future conversion costs.

The high investment in current software and the high cost of replacement often lead to prolonged use of software designs that have become obsolete. Obsolescence may be due to changing technology or economic factors, or to changes in organizations and work loads. Major software subsystems should be evaluated periodically, to identify modifications that would eliminate operational limitations, or would use

improved hardware or software technology. Major improvement in an existing system may be economically done through progressive stages of rework. Equally important, rework to standardize or restructure component routines may increase the feasibility of transfering the software directly to a future computer configuration. Such prospects empnasize the importance of standard documentation. Because the replacement of modules or interfaces would be impractical without clear definition of the existing software design.

*Use configuration management methods to control software status.

Configuration management involves close control of operational software modifications, to assure that continuing service is not adversely affected by changes and that the changes made are sufficiently important to justify their cost within the total maintenance effort available. Proposed changes should be agreed to jointly by the programming group and the users, and scheduled with the appropriate priority as part of the total workload of the programming team. Changes should first be made to a test copy of the software, not the same copy used for everyday service. Thus testing can proceed without delaying operational service, and only a fully checked out system is delivered for use.

*Periodically review the Project Record and current documentation, to insure that maintenance results are being recorded for later use by other programmers.

Maintenance responsibility is sometimes misconstrued as a personal prerogative that may be done with each programmer's style, unless management exerts its authority for quality control to meet the organization's needs. Documentation is a tedious but crucial chore, and it is inexcusable for operational software to be altered with no evidence except in the code itself. As in all phases of software work, management action should be as immediate as possible. The team review process is recommended for it may remove any appearance of management inquisition. Weekly meetings that include reviews of documentation updates may be sufficient to keep project information current and complete.

THE NEED FOR SOFTWARE PRODUCTION TOOLS

*Software tools are computer programs that automate tasks in the management, production, and testing of software.

A wide variety of tools have been developed and used in the software field, but few standard tools are recognized as applicable to every project. The most widely used tools are the compilers for high level programming languages, that transform program statements into executable machine codes. Many other tools can appreciably reduce the tedium of programming tasks and can expedite the application of standards to a programmer's products. Surveys of computer installations, as well as recent project experiences, show that the value of effective tools is not widely appreciated, and an adequate repertoire of effective tools is seldom provided. The importance of good software production tools is confirmed overwhelmingly by all recent experience and authoritative research in software engineering.

*Provide interactive computer support for programming on every project.

Developing programs and documentation on an interactive computer facility significantly improves the efficiency and working conditions for the programming team. Programmers can concentrate on technical problem solving without being limited and burdened by scheduling of computer runs, as in a batch computer support facility. Tentative design ideas can be experimentally programmed and evaluated in a short time, allowing the development of a quality design to move along rapidly. Various debugging and testing tools can be used to shorten the time for identifying sources of errors. A separate interactive support facility is advisable when the operational computer is unsuitable for interactive usage because it is intended for a batch or real-time application. The technology and economics of interactive support software have been amply proven, and the availability of basic systems is adequate for this general recommendation.

*Use preprocessors to supplement a standard high level programming language.

A preprocessor accepts programming statements written in an improved dialect of a standard programming language, and translates them into standard language acceptable to a compiler program. The merit of a preprocessor is that restrictive conventions that still exist in standard languages, such as card column orientation, can be

circumvented for improving programmer productivity and reducing errors, without losing the transportability advantages of the standard language. For example, structured programming could be applied even with FORTRAN, which is inherently unsuited for it. Equally important, many of the coding standards of a project could be checked for each program by a preprocessor, as a means of automated quality control. However, a preprocessor may be difficult to use together with a symbolic debugger, because the debugger output would refer to the preprocessor output statements, and so debugging results would be difficult to correlate with the source program statements.

*Establish a standard library of specialized software tools.

Software tools generally are inexpensive, often available from other users for the cost of reproduction, and sometimes concise and easily modified for unique project needs. Tools are usually specific to one programming language, and they perform a limited set of functions on programs written in that language. For each language in use then, a basic set of tools can be assembled, with each one assisting in an important, distinct design and programming task.

*Use an editor, program librarian, and program analyzer as standard tools.

An editor assists the programmer in making numerous changes and additions to a program, without the burden of filling out coding forms or manipulating format details, and with less risk of errors in repetitive, trivial changes. A good editor will automatically perform routine formatting and standardization actions, and with prompting and checking facilities included, it can prevent or identify trivial syntactic errors that a programmer may make.

A program librarian is a necessary complement to an editor, and provides for storing on the computer and retrieving all program modules and distinct versions of them that may develop during a project. The usual file system of an interactive computer facility may serve as an adequate program librarian. Features that are useful and often provided in special tools for this purpose include capabilities to time and date stamp different modules, to employ special naming conventions for programs, to efficiently store related program versions by storing only the relative changes, etc.

A program analyzer is software that scans the source statements in another program to collect information about

them or to provide for data collection about the program behavior when it is executed. A variety of analyzers may be useful, for special formatting of programs, for auditing adherence to coding standards, for extracting cross reference and data usage reports, for performance measurement, for test monitoring, and similar purposes.

Many tools of these types are available, and special tools are often economical to develop for a project.

*Use a simple project information system to maintain data on schedules, milestones, and resources used on distinct tasks and activities.

Simplicity is the key to keeping the management and administrative effort within bounds for small projects. For large projects, fiscal and contractual requirements may dictate extensive record keeping, but it is still advisable for technical management to have simplified project data that may expose basic problems, rather than to be buried under all possible details. If a computer information system is used, it should maintain a gross network of major milestones and produce a few brief reports on manpower and other expenditures for these milestones and other support activities.

STANDARDS FOR QUALITY CONTROL

A number of distinct quality control techniques are known to be effective, and some have been briefly described in the preceding. In all cases, there is no documented standard of practice for the technique, but there are published articles and sources of experience to help in developing appropriate standards for an interested organization. A summary definition of known techniques is given below.

PROJECT RECORD--An official journal of project plans, specifications, schedules, work assignments, budgets, technical standards, etc. that serves to inform all concerned parties of the goals, work methods, history, and

status of the effort.

PROJECT PLAN--A detailed plan of milestones, resource allocations, and quality control procedures to be used in the development phases for a software product. Incorporated in the Project Record and updated as needed during development.

DOCUMENTATION STANDARDS--Uniform requirements on documents for the project and the software, that define scope and identification of each document, as well as format and content, and that cover the principal specifications and technical descriptions, both internal to the project and deliverable with the software.

AUTOMATED DOCUMENTATION AIDS--Computer software for text editing and document formatting, and for automated preparation of program documentation such as flow charts or cross reference listings of data elements and program statements.

STRUCTURED SPECIFICATIONS--The approach of coordinating all specification documents in format and content, and using formal or mathematical specification techniques to reduce ambiguity. This facilitates successive validation milestones as increasingly detailed specifications are developed. The series of specifications involved begins with the feasibility study description of system concept, continues with the functional requirements, and extends to the individual program module specifications.

QUALITY and PERFORMANCE REQUIREMENTS--Augmentation of the functional requirements, to state the specific quality attributes expected of the software and the quantitative performance necessary in key operations or test cases.

SPECIFICATION CONTROL--A formal process of reviewing and approving proposed changes to specifications, with consideration of their technical validity, impact on project resources and schedule, and impact on work completed or in progress. The goal is to inhibit continuing, capricious changes, and instead stabilize the specifications on distinct deliverable versions of the software that can be managed for production.

STRUCTURED PROGRAMMING--Technical practices and criteria governing particularly the programming techniques and language usage for increasing programming productivity and reducing program errors.

TOP-DOWN PROGRAM DEVELOPMENT--The approach of implementing and testing program systems by building program modules starting with those at the most general, application-oriented level (the "top"), and proceeding successively down to the most specialized, computer-dependent level (the "bottom").

DESIGN and CODING STANDARDS--Detailed specifications of generally applicable design requirements based on quality objectives, including such programming conventions as indenting and spacing of program statements, use of comments, naming of variables, use of language features, etc.

SOURCE CODE CONTROL--The process of labeling and archiving program modules accepted for formal technical control, thereby limiting future programmer modifications to formally reviewed and approved changes.

TEAM REVIEWS--Technical team conferences to audit design proposals, specifications, and completed programs for quality.

UNIT FOLDERS--The programmer's technical records of design and testing work on individual program modules, sometimes called "unit development folders". As standard documents in a project, these augment the project record and specifications by providing more technical documentation as necessary for reviews and inspections. Especially important in large projects subject to frequent personnel changes or reassignments.

INSPECTIONS--Formal review and auditing of subsystems or groups of program modules at design, code, and test stages, by an independent technical team. Applicable to larger projects.

STANDARD PRODUCTION TOOLS--Computer software to aid the programming team in creation, documentation, testing, and analysis of the deliverable programs. These would include, for example, the standard programming language compiler, program librarian, test data generator, symbolic debugger, etc.

PROGRAM ANALYZERS--Software tools for automated analysis of program modules, to confirm complete adherence to coding standards and quality criteria, and to assure thorough testing according to standards.

PROJECT INFORMATION--Recording and analysis of the allocation and expenditure of team effort on different

modules and milestones for the program system, and on various production activities, in order to monitor progress and to support best practical resource management.

TESTING STANDARDS--Specific criteria governing the program testing to be performed, to assure that programs are uniformly tested by all programmers and that all program instructions and capabilities are tested to the extent practical.

INDEPENDENT TEST PLANS--Use of independent technical specialists to prepare comprehensive tests of the integrated software system.

FAULT REPORTING--A formal process for documenting observed errors or discrepancies in program operation and capability, to aid debugging and reliability assessment, and to validate software design.

CONFIGURATION MANAGEMENT--A formal process of controlling the operational software status and reviewing proposed improvements, to best direct maintenance effort and to minimize impact of maintenance and testing on operational service.

FITTING CONTROLS TO THE PROJECT SIZE

Software projects differ tremendously in complexity, acceptable schedule, and available resources. More than anything, the available personnel determine what technical goals are achievable and what quality controls may reasonably be imposed. Figure 6 illustrates a basic judgement of where the above controls are best utilized in regard to project size. The controls that are perceived to be more demanding of technical or supporting effort are not recommended for smaller projects.

Figure 6. Chart of Recommended Quality Controls

Quality Control	Development or Improvement Effort in Man-Years			
	Less than 1/2	1/2 to 2	2 to 5	Over 5
Project Record	X	X	X	X
Project Plan	X	X	X	X
Quality/Performance Requirements	X	X	X	X
Standard Documentation	X	X	X	X
Program Analyzers	X	X	X	X
Standard Production Tools	X	X	X	X
Structured Programming	X	X	X	X
Top-Down Development	X	X	X	X
Structured Specifications		X	X	X
Specification Control		X	X	X
Automated Documentation Aids		X	X	X
Design/Coding Standards		X	X	X
Team Reviews		X	X	X
Testing Standards		X	X	X
Project Information		X	X	X
Source Code Control			X	X
Unit Records			X	X
Fault Reporting			X	X
Configuration Management			X	X
Inspections				X
Independent Test Plans				X

FINAL HOMILIES

A few simple cautions may help to avoid unqualified failure in undertaking software development, given present knowledge of how best to do it.

Choose reasonable goals. Although computer technology has amazing capabilities, we remain primarily limited by human creativity in regard to software achievements.

Have a detailed plan. Important tasks may be forgotten or expectations may expand unless all is recorded as a reminder.

Put one person in control. There should be strong leadership toward one well understood concept.

Use good tools. Productive tools are the only certain way to accelerate human accomplishments.

Cater to the software user. The user judgement is the ultimate answer on success or failure of the product.

Build up in steps. Confidence from successful achievement of modest goals will lead more quickly to grander objectives.

Admit mistakes early. Pursuit of the hopeless leads only to further despair--poor choices and misconceptions require drastic remedies as early as possible.

Consider starting over. Patching a poor result has limited return; a wiser action may be to start fresh at the beginning.

REFERENCES

Baker, F. T., "Structured Programming in a Production Programming Environment", Proceedings of the International Conference on Reliable Software, 21-23 April 1975, Los Angeles, pp172-185, IEEE Computer Society, Long Beach, California.

Brooks, F. P., Jr., The Mythical Man-Month: Essays on Software Engineering, Addison-Wesley Publishing Co., Reading, Mass., 1975, 195p.

Comptroller General of the United States, Lessons Learned About Acquiring Financial Management and Other Information Systems, August 1976, 61p, available from U. S. Government Printing Office, Washington, D. C. 20402, Stock No. 020-000-00138-1.

Endres, A., "An Analysis of Errors and Their Causes in System Programs", IEEE Transactions on Software Engineering, v. SE-1, n. 2, June 1975, 140-149, IEEE Computer Society, Long Beach, Calif.

Fagan, M. E., "Design and Code Inspections to Reduce Errors in Program Development", IBM Systems Journal, v. 15, n. 3, 1976, 182-211.

Hecht, Herbert, Measurement, Estimation, and Prediction of Software Reliability, NASA CR-145135, Aerospace Corporation, El Segundo, Calif., January 1977, 41p.

IBM Corp., Data Processing Division, Improved Programming Technologies--An Overview, Installation Management Report GC20-1850-0, October 1974, 19p.

IBM Corp., Data Processing Division, HIPO--A Design Aid and Documentation Technique, Installation Management Report GC20-1851-0, October 1974, 130p.

Kernighan, B. W. and Plauger, P. J., The Elements of Programming Style, McGraw-Hill Book Company, New York, 1974, 147p.

Kernighan, B. W. and Plauger, P. J., Software Tools, Addison-Wesley Publishing Co., Reading, Mass., 1976, 338p.

Mills, Harlan, "Software Development", IEEE Transactions on Software Engineering, December 1976, 265-273.

Myers, Glenford J., Software Reliability: Principles and Practices, John Wiley and Sons, New York, 1976, 360p.

National Bureau of Standards, Guidelines for Documentation of Computer Programs and Automated Data Systems, Federal Information Processing Standards Publication 38, 1976 February 15, 55p.

Rochkind, M. J., "The Source Code Control System", Proceedings of the First National Conference on Software Engineering, 11-12 September, 1975, Washington, D. C., pp37-43, IEEE Computer Society, Long Beach, Calif.

Rubey, R. J., et al., "Quantitative Aspects of Software Validation", IEEE Transactions on Software Engineering, v. SE-1, n.2, June 1975, 150-155, IEEE Computer Society, Long Beach, Calif.

Van Tassel, D., Program Style, Design, Efficiency, Debugging, and Testing, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1974, 256p.

Walston, C. E. and C. P. Felix, "A Method of Program Estimation and Measurement", IBM Systems Journal, v. 16, n. 1, 1977, 54-73.

Yourdon, E., Techniques of Program Structure and Design, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1975, 364p.

Yourdon, E., "A Case Study in Structured Programming: Redesign of a Payroll System", Digest of Papers of IEEE COMPCON, September 9-11, 1975, Washington, D. C., 119-122, IEEE Computer Society, Long Beach, Calif.

NBS-114A (REV. 7-73)

U.S. DEPT. OF COMM. BIBLIOGRAPHIC DATA SHEET	1. PUBLICATION OR REPORT NO. NBS SP-500-11	2. Gov't Accession No.	3. Recipient's Accession No.

4. TITLE AND SUBTITLE

COMPUTER SCIENCE & TECHNOLOGY:

Computer Software Management: A Primer for Project Management and Quality Control

5. Publication Date July 1977

6. Performing Organization Code

7. AUTHOR(S) Dennis W. Fife

8. Performing Organ. Report No.

9. PERFORMING ORGANIZATION NAME AND ADDRESS

NATIONAL BUREAU OF STANDARDS
DEPARTMENT OF COMMERCE
WASHINGTON, D.C. 20234

10. Project/Task/Work Unit No.

11. Contract/Grant No.

12. Sponsoring Organization Name and Complete Address *(Street, City, State, ZIP)*

Same as Item 9

13. Type of Report & Period Covered

14. Sponsoring Agency Code

15. SUPPLEMENTARY NOTES

Library of Congress Catalog Card Number: 77-608127

16. ABSTRACT *(A 200-word or less factual summary of most significant information. If document includes a significant bibliography or literature survey, mention it here.)*

Today, providing computer software involves greater cost and risk than providing computer equipment, because hardware is mass produced by industry using proven technology, while software is still produced mostly by the craft of individual computer programmers. This brief guide is intended for managers who are responsible for computer projects, to explain the use of quality controls and software management methods. The typical problems of software development are explained. Over twenty distinct quality controls are defined, and recommendations are given for software management actions. Empirical information is included that would help top executives to appreciate the potential problems and importance of software management.

17. KEY WORDS *(six to twelve entries; alphabetical order; capitalize only the first letter of the first key word unless a proper name; separated by semicolons)*

Computer management; computer programming; computer project control; computer software; software engineering; software quality; software reliability.

18. AVAILABILITY

- [x] Unlimited
- [ ] For Official Distribution. Do Not Release to NTIS
- [x] Order From Sup. of Doc., U.S. Government Printing Office Washington, D.C. 20402, SD Cat. No. C13.10:500-11
- [ ] Order From National Technical Information Service (NTIS) Springfield, Virginia 22151

19. SECURITY CLASS (THIS REPORT)	21. NO. OF PAGES
UNCLASSIFIED	58
20. SECURITY CLASS (THIS PAGE)	**22. Price**
UNCLASSIFIED	

USCOMM-DC 29042-P74